Proceedings 12th IEEE European Test Symposium, Freiburg, Germany, May 2007

An Integrated Built-in Test and Repair Approach for Memories with 2D Redundancy

Philipp Öhler and Sybille Hellebrand
University of Paderborn Germany
{oehler,hellebrand}@uni-paderborn.de

Hans-Joachim Wunderlich
University of Stuttgart Germany
wu@informatik.uni-stuttgart.de

Abstract

An efficient on-chip infrastructure for memory test and repair is crucial to enhance yield and availability of SoCs. Therefore embedded memories are commonly equipped with spare rows and columns (2D redundancy). To avoid the storage of large failure bitmaps needed by classical algorithms for offline repair analysis, existing heuristics for built-in repair analysis (BIRA) either follow very simple search strategies or restrict the search to smaller local bitmaps. Exact BIRA algorithms work with sub analyzers for each possible repair combination. While a parallel implementation suffers from a high hardware overhead, a serial implementation leads to high test times. The integrated built-in test and repair approach proposed in this paper interleaves test and repair analysis and supports an exact solution without failure bitmap. The basic search procedure is combined with an efficient technique to continuously reduce the problem complexity and keep the test and analysis time low.

1. Introduction

State of the art systems-on-a-chip (SoCs) typically devote a large percentage of the chip area to various kinds of memory cores. According to the International Roadmap for Semiconductors (ITRS) the percentage of memory in SoCs will continue to increase rapidly [7]. At the same time the shrinking feature sizes will lead to increasing parameter variations and a high susceptibility to defects. As memories are traditionally designed with more aggressive design rules than logic cores, they play a crucial role for the yield and reliability of a SoC. Embedding the necessary infrastructure for a built-in test and repair is essential to achieve acceptable yields and to guarantee a satisfactory availability in the field [19, 20].

Memory repair relies on spare elements at different levels of the design hierarchy. The most common form is 2D redundancy where both spare rows and spare columns are added to the memory [1, 2, 6, 8-11, 13-18]. With increasing defect rates the number of spare parts must be increased to keep the desired yield. Furthermore the possibility for on-line repair in the field becomes more and more important to compensate new defects during the lifetime of a system. Thus an optimal repair solution with a minimum number of spares is preferred to provide a good basis for future repairs.

Usually the repair process for 2D redundancy consists of several steps. First the memory is tested, and the information about faulty elements is collected in a *failure bitmap*. Then *repair analysis* attempts to find an allocation of spare elements, such that all faults are covered at minimum cost. As a result either the memory is identified as not repairable or a *repair signature* is obtained which is the basis for soft or hard repair.

Strategies for 2D repair analysis have been investigated for more than two decades. However, the classical approaches have been developed for offline test and repair analysis and cannot be directly applied on chip [3, 9, 15, 18]. Nevertheless, they provide the foundation for *built-in repair analysis* (BIRA). In particular, Kuo and Fuchs have shown that the problem of optimal 2D redundancy allocation is NP-complete, and they have also proposed a systematic branch and bound approach based on a binary search tree [9].

Adapting these techniques to a fully built-in repair analysis poses two problems. Firstly, most of the search strategies rely on a complete failure bitmap. Secondly, the inherent data structures to organize the search can grow very large. To overcome these problems, most approaches for built-in repair analysis either follow only very simple search strategies, partition the memory into smaller parts, or they rely on local failure bitmaps [1, 6, 13, 16]. With CRESTA Kawagoe et al. have proposed a pioneering BIRA approach, which guarantees to find the optimal solution [8]. Similarly as the early work in [9] it is based on a

This work was supported by the DFG-grants HE 1686/2-1 and WU 245/4-1

binary search tree, but a separate *sub analyzer* is implemented for each path in the tree. This way all possible solutions can be analyzed in parallel. However, the hardware cost grows rapidly with the number of redundancies. For a memory with r redundant rows and c redundant columns $b(r + c, r)$ sub analyzers are needed, where $b(n, k)$ denotes the binomial coefficient n over k. A serial processing of the $b(r + c, r)$ sub analysis tasks as mentioned in [14] can reduce the hardware cost, but leads to very high test and analysis times.

The integrated built-in test and repair approach proposed below performs repair analysis concurrently with test application. This way an optimal solution can be found without any failure bitmap. The basic algorithm uses a binary search tree and is implemented with a stack of size $r + c$. It can be combined with a strategy to continuously reduce the problem complexity. Two small content addressable memories (CAMs) with only $2r{\cdot}c$ entries, each, support the detection of necessary repairs and the fast identification of non-repairable memories.

2. Basic concepts

As the test and repair scheme proposed in this paper uses a binary search tree, the basic concepts and principles introduced in [9] are briefly summarized with the help of the small example memory of Figure 1.

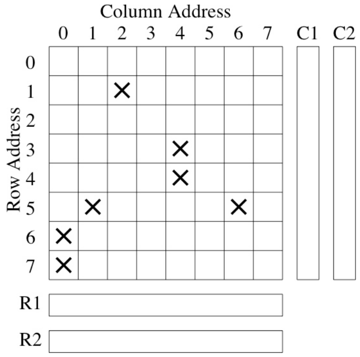

Figure 1. Example memory with 2x2 spares.

Each node in the search tree of Figure 2 corresponds to repair decision “row repair” or “column repair” for a fault in the memory. As long as the tree is under construction, a node is called “closed”, if all decisions starting from this node have already been explored, else it is called “open”. A leaf is reached when a successful repair scheme is found or no more repair resources are available. In case of a successful repair, the path from the root to the leaf provides the repair signature. To facilitate a continuation of the search from an arbitrary open node, for each node the partial repair configuration corresponding to this node is attached, i.e. the addresses of rows and columns to be replaced by spare parts are listed. The highlighted path in Figure 2 leads to a successful repair configuration, and there are still several open nodes, which can be explored to improve the solution.

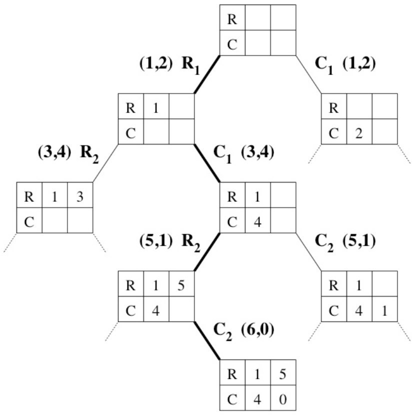

Figure 2. Partial search tree.

For a memory with r redundant rows and c redundant columns the maximum height of the tree, i.e. the maximum length of a path from the root to a leaf node, is $r + c$. The leaf nodes in a complete search tree correspond to all solutions exploiting all resources. As there are r rows distributed among $r + c$ spares, there are $b(r + c, r)$ leaf nodes in a tree enumerating all possible repair configurations. To reduce the size of the search tree Kuo and Fuchs propose to perform a “must repair” phase before starting the binary search [9].

Observation 1 (“must repair”): For a memory with r redundant rows and c redundant columns the following repair decisions are mandatory: If there are more than c faults in a row, then there are not enough columns to cover all the faults, and a row must be selected for repair. Similarly, more than r faults in a column require a column repair.

3. Integrated built-in test and repair

3.1. The basic algorithm

The integrated test and repair approach introduced in this section builds the search tree concurrently with the test of the memory and avoids the need for large failure bitmaps. Whenever a new fault is detected during test, a (preliminary) repair decision is made and a new node is added to the search tree. If backtracking from node w to v in the search tree is necessary, the preliminary repair decisions between the two nodes must be cancelled and the test must be restarted with the (partial) repair signature corresponding to node v.

For a low cost hardware implementation the search is organized as a “depth-first” traversal, as it can be implemented using a stack, which is limited by the height of the search tree [5]. Figure 3 shows the resulting on-chip infrastructure in more detail.

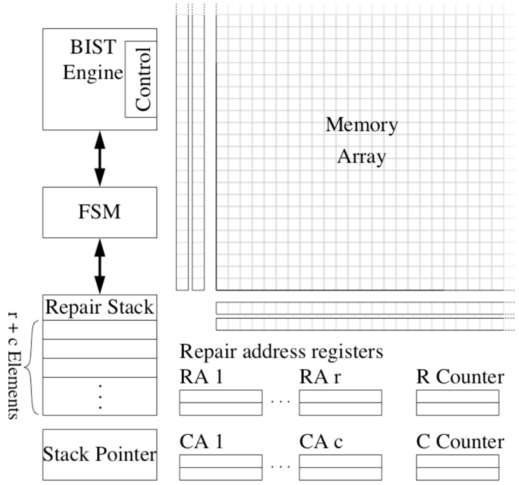

Figure 3. On-chip infrastructure.

The BIST engine contains address generators and other resources to implement the memory test relying on wellknown techniques [4]. Its control part must be adapted, such that communication with the FSM controlling the repair analysis is possible. The repair stack stores all the necessary information about the repair configuration currently being explored. It contains at most $r + c$ records describing the repair type and the status of a node. The repair type is encoded by two bits: ‘00’ represents “no repair”, ‘10’ stands for “row repair”, and ‘01’ for “column repair”. The status is also described by a 2-bit code. If it is ‘11’, then the node is closed, else the node is open. The status code ‘00’ indicates that no successor has been generated yet, ‘10’ (‘01’) shows that a row (column) repair has already been explored. It is not necessary to store the complete repair configuration with each node on the stack. The row and column counters as well as the repair address registers in Figure 3 are sufficient to keep track of the assignments made during search and to store the best repair configuration found so far. The counters indicate how many spares have already been allocated and the repair address registers attach the address of the faulty row or column to the allocated spare element.

The search starts with resetting the repair registers and the repair counters. Furthermore, the root node is pushed on the stack with repair code ‘00’ and status code ‘00’. Then during each step of the search the node on top of the stack is analyzed. If the node is already closed, then it is popped off the stack, i.e. backtracking is started. If the node is open, then the next repair decision depends on the status code, the availability of resources, and the repair strategy followed (e.g. “row first”, “column first”, “random” or “balance remaining resources” [12]).

Whenever a new node is pushed on the stack, then the address of the faulty row or column is stored in the address register of the first available redundant row (column), and the row (column) counter is updated. Since the spare rows and columns are used in a fixed order during the search, canceling repair decisions during backtracking simply corresponds to decrementing the row or column counter. To continue the search after backtracking, the test must be restarted with the partial repair signature derived from the valid addresses in the repair registers.

If the test finishes during search, then the contents of the stack corresponds to a successful repair configuration. Once the first solution has been found, the number of spare elements in the best solution so far provides a criterion to prune the search tree. Sub trees corresponding to solutions with the same or more elements can be cut off. The search stops when the stack is empty and all alternatives have been explored. If the test still detects additional faults at this point, then the memory is not repairable.

The complete test and repair process is illustrated for the small example memory of Figure 1. Here it is assumed that a “row first” strategy is followed. If the faults are detected in the order (1, 2), (3, 4), (4, 4), (5, 1), (5, 6), (6, 0), and (7, 0), then the search tree of Figure 4 is obtained, the nodes of which are labeled in the order of traversal.

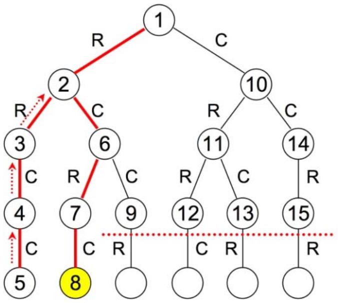

Figure 4. Depth-first traversal.

Figure 5 shows the stack after the traversal of the first path with nodes ‘1’ to ‘5’. The repair codes indicate that two row repairs and two column repairs have been performed starting from the root node ‘1’. The addresses of the replaced elements are listed in the repair registers. The status codes show that from nodes ‘1’ and ‘2’ further alternatives can be explored while nodes ‘3’, ‘4’ and ‘5’ are already closed. This is due to the fact that there are no more appropriate resources to continue the search from these nodes. With node number '5' on top of the stack a dead-end is reached, since all spare elements have been used and the test still detects another fault at address (5, 6).

Node Number	*Repair Stack*	
	Repair Code	Status Code
1	00	10
2	10	10
3	10	11
4	01	11
5	01	11

Repair Registers	
Spare	*Address*
R1	1
R2	3
C1	4
C2	1

Figure 5. Repair stack for the first path.

Backtracking starts and continues until the open node '2' is reached. The last three repair steps have been cancelled by decrementing the repair counters accordingly. The test is restarted with the partial repair signature corresponding to node '2' and finds the first fault at address (3, 4). The status of node '2' indicates that only a column repair is left as an unexplored alternative. When node number '8' is reached, a first solution of the repair problem is found. The search continues until the stack is empty and the completed search tree is traversed. But in this example no other solutions for the repair problem can be found.

The proposed scheme guarantees to find the best repair solution with a very simple algorithm. It is therefore referred to as *basicSolve*. However, as already pointed out, backtracking in the search tree implies a restart of the test, which may lead to high test and repair times. To overcome these problems an efficient strategy to reduce the search space is presented in the next section.

3.2. Continuous reduction of the search space

The must repair criterion stated in Observation 1 provides the basis for the continuous reduction of the search space presented in this section. However, it is not restricted to a single preprocessing step as in many other BIRA approaches. As each repair changes the number of available spares, new situations fulfilling the must repair criterion may occur after a repair step. In particular, after a must repair step other must repair decisions may become necessary. Therefore, a dynamic must repair analysis is performed as a preprocessing step and after each repair node pushed on the stack.

For an efficient hardware implementation an intelligent fault list is used to manage fault addresses. This fault list consists of a table for row addresses and a table for column addresses, each of which is realized by a small CAM of size $2r{\cdot}c$. During memory test, each detected fault address is compared against the contents of the two CAMs. As a result the row CAM provides the number of faults with the same row address, and the column CAM provides the number of faults with the same column address already stored.

If the number of faults with the same row address has already reached c, then, with the new fault, the must repair criterion for a row repair is fulfilled. Similarly, if the number of faults with the same column address is r, this implies a mandatory column repair. After performing the corresponding repair, a dynamic must repair analysis is carried out for all faults stored in the CAM with the updated values for r and c.

If neither a row nor a column must repair can be triggered, the row address of the new fault is stored in the row CAM, and the column address is entered in the column CAM. The maximum number of fault addresses which can be collected without invoking a must repair is $2r{\cdot}c$ [6]. Therefore it is sufficient to select $2r{\cdot}c$ as the CAM size for both the row and the column CAM. This observation is also useful for pruning the search tree and for the early identification of non-repairable memories. If both CAMs are full, and a new fault is detected without leading to a must repair, then the memory is proven to be non-repairable and the search can be stopped.

The basic algorithm of Section 3.1 combined with the proposed strategy is called *intelligentSolve*. Its flow is illustrated again for the small example of Figure 1 with faults detected in the same order as before. After pushing the root node on the stack the test is started and the fault list shown in Figure 6 is collected.

Fault Number	*Fault Address*	
	Row	*Column*
1	1	2
2	3	4
3	4	4
4	5	1
6	5	6
7	7	0

Figure 6. Fault list captured during test.

As a reduction based on the must repair criterion is not possible, and the maximum size of the CAMs has not been exceeded, the first repair node is generated exploring a row repair. This eliminates the first entry in both tables, and the number of available spare rows is decremented. The following must repair analysis with the updated values for r and c immediately identifies two must repairs for column addresses 4 and 0. After the corresponding repairs, c is updated to zero, and a must repair situation is found for row address 5. At this point a first solution of the repair problem is found with only two nodes on the stack and dynamic must repair.

To improve the solution the search is continued by backtracking to the root node. This time a column repair is explored to cover the fault at address (1, 2). This decision invokes a must repair for row 5, which in turn leads to a must repair of column 0. At this point the must repair criterion is fulfilled for column number 4, but there is only one spare row left. Hence the search for the optimal solution can be stopped after backtracking only once.

4. Experimental results

To evaluate the proposed strategies, the algorithms *basicSolve*, *intelligentSolve* and a version of *intelligentSolve* stopping at the first solution (*intelligentSolveFirst*) have been simulated for a 1024×1024-bit memory using the "row first" strategy. The experiments have been performed for different redundancy configurations and for different numbers of random defects. The redundancy configurations varied from two spare rows and two spare columns (2×2) to five spare rows and five spare columns (5×5). The number of random defects has been linearly increased ranging from one to fifteen. A random defect can result in a single faulty cell, a faulty row or column, a "line fault" consisting of several adjacent faulty cells in a row or column, or a cluster fault affecting up to 3×3 cells. The considered distributions of defect types are listed in Table 1.

Table 1: Distribution of Defect Types.

Defects	*Distributions*		
	d_1	d_2	d_3
Row	0.10	0.10	0.10
Column	0.10	0.10	0.10
Line Fault	0.10	0.20	0.40
Cluster	0.05	0.10	0.20
Single Cell	0.65	0.50	0.20

Each experiment has been repeated 1000 times with randomly generated addresses of the faulty locations. Since all redundancy configurations and all the three defect distributions show similar trends, only the results for d_2 for a 5×5 configuration are discussed in the following. The average results for the number of test restarts are illustrated in Figure 7. In the grey area all experiments ended up with non-repairable memories. The curves in Figure 7 show that the dynamic must repair proposed in Section 3.2 greatly reduces the search complexity. In particular, the procedure *intelligentSolveFirst* does not require any backtracks or only very few backtracks in the region where a high repair rate is possible.

However, each entry in Figure 7 only shows the mean value for the results of 1000 random experiments. To get deeper insight into the behavior of the repair algorithm, Figure 8 provides a histogram analyzing the detailed results for the case where *intelligentSolve* reaches the maximum average value of 77.685 backtracks.

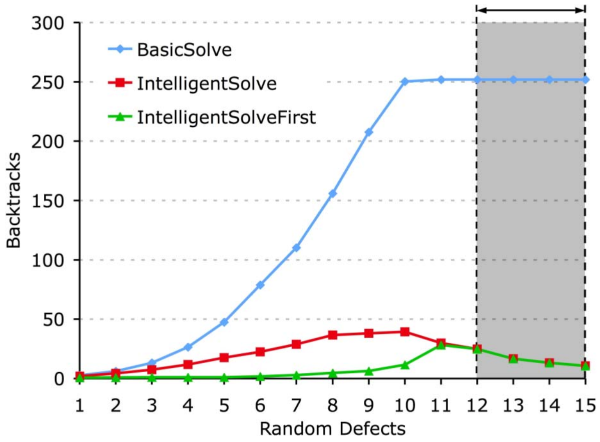

Figure 7. Restarts for 5×5 redundancy.

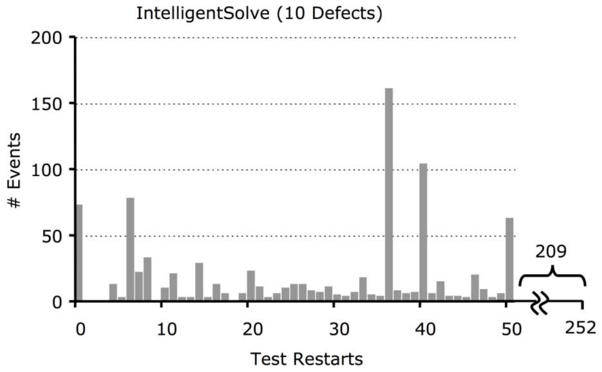

Figure 8: Histogram for 10 random defects.

It can be observed that the peaks are found at much lower values than the average value suggests. This shows that the repair problem can be solved with a moderate number of backtracks in many cases. In 30% of the experiments a repair solution was found with less than 20 restarts, and in only 20% of the cases more than 50 restarts were needed. Furthermore, in this example, where the high number of defects makes repair very difficult, the early abort criterion is already very powerful. In more than 70 cases the memory can be identified as non-repairable in the preprocessing phase.

As pointed out above, the test and repair time can even be reduced further using the *intelligentSolveFirst* algorithm, which finds a solution without any backtracks in many cases. In this case an optimum solution can no longer be guaranteed, but it is interesting to note that the quality of the results differs only slightly for *intelligentSolve* and *intelligentSolveFirst*. The average number of additional spares required by *intelligentSolveFirst* never exceeded 20% of the optimal solution determined by *intelligentSolve*. For hard to repair memories with a high number of defects the additional average cost was even below 10 %. which corresponds to at most 1 additional spare part.

Thus *intelligentSolveFirst* provides an excellent solution, if the redundancy configuration is well adapted to the expected defect distribution and a successful repair can be expected with a high probability. As the proposed approach guarantees to find a solution, if the memory is repairable, a comparison to the repair rates of other approaches is omitted. Comparing hardware cost as well as test and repair time shows the following facts. For a 5×5 configuration CRESTA already needs $b(10, 5) = 252$ sub analyzers to be implemented in hardware compared to a stack of maximum size 10, two CAMs with 50 entries, each, and a small FSM. Processing the sub analysis tasks serially as mentioned in [14] would require $b(10, 5) = 252$ restarts of the test, while the proposed approach needs less than 50 restarts in the majority of cases. The local bitmap proposed in [6] contains the same information as the proposed intelligent fault list, but it has a size of $(r(c + 1) + r) \cdot (c(r + 1) + c) = 35 \cdot 35 = 1225$ bits for the 5×5 configuration.

5. Conclusions

The integrated built-in test and repair approach proposed in this paper supports a low cost hardware implementation by interleaving test and repair analysis. A stack of size $r + c$ and small FSM are sufficient to realize the basic depth-first algorithm. Adding two small CAMs of size $2r\cdot c$, each, allows a continuous reduction of the search space, and thus a reduced number of backtracks and restarts of the test. In particular the procedure *intelligentSolveFirst* provides a very efficient solution for repairable memories, because it achieves high quality results with less hardware and shorter test and analysis times than other state of the art BIRA schemes.

6. References

[1] D. K. Bhavsar, ”An algorithm for row-column self-repair of RAMs and its implementation in the Alpha 21264.” Proc. IEEE Int. Test Conf. (ITC), Atlantic City, NJ, USA, pp. 311-318, Sept. 1999.

[2] K. Chakraborty et al., “A physical design tool for built-in self-repairable RAMs.” IEEE Trans. on VLSI Systems, Vol. 9, No. 2, pp. 352-364, April 2001.

[3] J. R. Day, “A fault-driven comprehensive redundancy allocation algorithm.” IEEE Design & Test of Computers, Vol. 2, No. 3, pp. 35-44, June 1985.

[4] S. Hamdioui, G. Gaydadjiev, and A. J. van de Goor, “The state-of-art and future trends in testing embedded memories.” Records Int. Workshop on Memory Technology, Design and Testing (MTDT’04). 2004.

[5] E. Horowitz, S. Sahni, S. Rajasekaran, “Computer Algorithms in C++.” New York: Computer Science Press, 1998 (2nd printing).

[6] C.-T. Huang et al., “Built-in redundancy analysis for memory yield improvement.” IEEE Trans. on Reliability, Vol. 52, No. 4, pp. 386–399. Dec. 2003.

[7] International Technology Roadmap for Semiconductors, 2005 Edition, http://www.itrs.net/.

[8] T. Kawagoe et al., “A built-in self-repair analyzer (CRESTA) for embedded DRAMs.” Proc. IEEE Int. Test Conf. (ITC), Atlantic City, NJ, USA, pp. 567-574, Oct. 2000.

[9] S.-Y. Kuo and W. K. Fuchs, “Efficient spare allocation in reconfigurable arrays.” Proc. 23rd ACM/IEEE Design Automation Conf. (DAC), Las Vegas, NV, USA, pp. 385–390, June 1986.

[10] J.-F. Li et al., “A built-in self-repair design for RAMs with 2-D redundancy.” IEEE Trans. on VLSI Systems, Vol. 13, No. 6, pp. 742–745, June 2005.

[11] S. Nakahara et al., “Built-in self-test for GHz embedded SRAMs using flexible pattern generator and new repair algorithm.” Proc. IEEE Int. Test Conf. (ITC), Atlantic City, NJ, USA, pp. 301-307, Sept. 1999.

[12] P. Oehler, S. Hellebrand, and H.-J. Wunderlich, “Analyzing Test and Repair Times for 2D Integrated Memory Built-in Test and Repair.” Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Krakow, Poland, April 2007.

[13] A. Seghal et al., “Yield Analysis for Repairable Embedded Memories.” Proc. IEEE European Test Workshop, Maastricht, NL, pp. 35-40, May 2003.

[14] S. Shoukourian, V. Vardanian, and Y. Zorian, “An Approach for Evaluation of Redundancy Analysis Algorithms.” Proc. IEEE Memory Technology, Design and Testing Workshop (MTDT’01), San Jose, CA, USA, pp. 51-55, Aug. 2001.

[15] M. Tarr, D. Boudreau, and R. Murphy, “Defect analysis system speeds test and repair of redundant memories.” Electronics, pp. 175-179, Jan. 12, 1984.

[16] T.-W. Tseng, J.-F. Li, and D.-M. Chang, “A built-in redundancy-analysis scheme for RAMs with 2D redundancy using 1D local bitmap,” Proc. Design, Automation and Test in Europe (DATE), Munich, Germany, pp. 53-58, March 2006.

[17] O. Wada et al., “Post-packaging auto repair techniques for fast row cycle embedded DRAM.” Proc. IEEE Int. Test Conf. (ITC), Charlotte. NC, USA, pp. 1016-1023, Oct. 2004.

[18] C.-L. Wey and F. Lombardi, “On the repair of redundant RAMs.” IEEE Trans. on CAD, Vol. 6, No. 2, pp. 222–231, March 1987.

[19] Y. Zorian, „Embedded memory test & repair: infrastructure IP for SoC yield.“ Proc. IEEE Int. Test Conf. (ITC), Baltimore, MD, USA, pp. 340-349, Oct. 2002.

[20] Y. Zorian, S. Shoukourian, “Embedded-Memory Test and Repair: Infrastructure IP for SoC Yield.” IEEE Design & Test, Vol. 20, No. 3, pp. 58-66, May/June 2003.